ChatGPT

High-Performance Graph Evaluation in Rust vs C++

Introduction

Implementing a computation graph with *extremely* high performance in Rust requires a careful design that avoids any unnecessary overhead. In C++, one can give each node direct pointers to its dependencies' values, making recomputation very fast. We want a similar setup in Rust – meaning at evaluation time we should **not** be doing string lookups, hash map queries, or other expensive indirections. Each node's operation is very small (simple arithmetic), so even minor overheads (function call indirection, pointer chasing, etc.) would dominate the cost. The goal is to design the graph such that evaluating it is essentially just straight-line numeric calculations, as close to C++ efficiency as possible.

Design assumptions/requirements: (as confirmed by the question context) - **Static graph** – Once the graph is built, its structure doesn't change during evaluation. This allows us to use fixed references or indices to link nodes. - **Unsafe optimizations allowed** – We can employ low-level techniques (raw pointers, unchecked access) for speed, provided we uphold safety invariants. - **Single enum for node types** – We will represent all operation types in one enum (closed set of node variants). This avoids dynamic dispatch overhead and allows static dispatch optimizations [1] [2]. - **Minimal maintenance for new nodes** – Adding a new node type should require updating *only* this one enum (and its match logic) – no scattered changes in multiple places.

With these in mind, we can approach a Rust design that matches the performance of the C++ approach.

Avoiding Common Pitfalls in Rust

A naïve implementation of a graph in Rust might use trait objects or hash maps for node values, but those approaches carry significant overhead: - **Dynamic dispatch and boxing:** Using a `Box<dyn Node>` for each node (trait objects) means each evaluation would involve a virtual function call (vtable lookup) and no inlining. This overhead is small per call, but in a hot loop of thousands of tiny operations it adds up. In fact, switching from static dispatch (e.g. an `enum` match) to dynamic dispatch incurs a performance penalty [1]. In cases where each operation is trivial, dynamic dispatch overhead becomes significant. It's well-noted that using an enum to dispatch (static dispatch) can be *much* faster than dynamic dispatch – often an order of magnitude difference [2]. Moreover, boxing each node spreads them out in heap memory, hurting cache locality. - **Hash maps or string keys:** Storing node outputs in a `HashMap` (keyed by node ID or name) or referencing inputs by strings is far too slow. A hash map lookup per node is expensive in terms of CPU and memory access (and string operations are even worse). One Rust developer noted that using an indirection via maps for node storage adds painful overhead – "the extra level of indirection from constantly doing table lookups is also a pain" [3]. Hash maps involve unpredictable memory access and hashing, which we cannot afford for each arithmetic operation. We need O(1) direct access to values. - **Pointer-chasing and branch misprediction:** If the graph is implemented with nodes pointing to each other via pointers (e.g. a linked structure), evaluating it means jumping around in memory following pointers. Such data-dependent memory access can wreak havoc on performance due to cache misses and branch mispredictions [4]. We

want to access memory in a predictable, linear pattern as much as possible, to leverage caches. Similarly, we prefer a tight loop with simple arithmetic and predictable branches. Every unpredictable branch or memory access (a "data-dependent load") can stall the CPU pipeline waiting for memory [4]. This is why we will organize data contiguously and avoid per-node dynamic decisions beyond a simple enum match.

In summary, we should **avoid**: dynamic trait calls, per-node allocations, strings or map lookups, and scatter-gun memory layout. All these would make evaluation "much too slow" given the high performance requirements.

Using an Enum for Node Operations (Static Dispatch)

To eliminate virtual call overhead, we use a single enum type to represent all possible node operation kinds. For example:

```
enum NodeOp {
    Constant(f64),              // stores a constant value
    Add { a: usize, b: usize }, // add values from node a and b
    Mul { a: usize, b: usize }, // multiply values from node a and b
    // ... other operation variants ...
}
```

Each node in the graph will have a `NodeOp` that encodes what computation it performs and the indices of its operand nodes (if any). This design has a closed set of node types known at compile time. The evaluation can then use a `match` on the `NodeOp` variant to execute the correct operation. This is a form of static dispatch – the compiler knows all variants and can optimize the dispatch (e.g. jump tables or inlined branches) [5]. In contrast to trait object calls, pattern-matching on an enum is **much faster** because it avoids dynamic dispatch overhead [2]. Adding a new variant (new node type) requires updating the enum and the `match` logic in the evaluator, but only in that one place (which the question asker deemed acceptable).

Why an enum dispatch is fast: With optimizations, the Rust compiler will often turn a match on an enum into a direct jump or a series of predictable branches. If there are many variants, it might use a jump table. Even in the worst case, a branch misprediction penalty is typically on the order of ~10-20 nanoseconds. By contrast, a dynamic dispatch not only risks branch mispredict but also prevents inlining and may involve pointer dereferences – easily costing significantly more. One source measured enum (pattern match) dispatch to be several times faster than `dyn Trait` calls in hot loops [2]. Thus, for our use case, the enum approach is the way to go.

Contiguous Storage for Nodes and Values

To mimic C++-style "direct references" to dependency values, we will store all node outputs in a contiguous array (for example, a `Vec<f64>`). Each node's inputs can then be referenced by the index into this array. This gives us constant-time, direct access to any dependency's value by index, without any hashing or string matching. Moreover, an index is essentially an offset – very cache-friendly if iterated in order.

Graph construction: We can accumulate nodes in a `Vec<NodeOp>` as we build the graph. Each time we add a node, we assign it an index (e.g., the position in the vector). We ensure that if node X depends on Y, then Y's index is lower than X's index – this is naturally achieved if we build in topological order (i.e. always add dependencies before the node that uses them). If the graph is defined in an arbitrary order, we can topologically sort it before evaluation.

Evaluation strategy: Once nodes and their `NodeOp` are set, we allocate a `values: Vec<f64>` of length equal to the number of nodes. The evaluation is then a simple loop from index 0 to N-1:

```
for i in 0..nodes.len() {
    values[i] = match nodes[i] {
        NodeOp::Constant(val) => val,
        NodeOp::Add { a, b }   => values[a] + values[b],
        NodeOp::Mul { a, b }   => values[a] * values[b],
        // ... other operations ...
    };
}
```

Each iteration performs one match and a couple of basic arithmetic operations or memory loads. There are no string keys, no hash maps – just direct index addressing. This is very efficient: the array access by index is an O(1) direct memory operation (which the CPU can often do in a single instruction). We also expect good locality: since we iterate sequentially and each node mostly reads values of recent (lower index) nodes, those values are likely still in cache. By keeping all values in one vector and all node definitions in another vector (or parallel to it), we utilize contiguous memory, which modern hardware prefetches and caches effectively [6].

Importantly, by structuring the evaluation as a straight loop, we minimize branch unpredictability. The only branch is the match on the node's variant. Even if the sequence of variants is somewhat random, the cost of a misprediction there is relatively small compared to, say, a cache miss. And because the operations themselves are so cheap, this single branch per node is acceptable. (If it became a concern, one could even experiment with grouping nodes by type to improve branch predictability, but that complicates execution order. In practice, the simple loop is usually fine.)

No data-dependent lookups: Using indices ensures that accessing an input value is just `values[index]`. The index is just a number stored in the node struct, so retrieving the value is a direct memory load from a known offset (plus a bounds check in safe Rust, which the compiler might optimize out knowing the index < current i). This avoids *data-dependent* pointer chasing. In other words, we don't, for example, fetch a pointer from a node, then follow it to another memory location far away; we just compute an array offset. By design, this greatly reduces cache-miss overhead. In high-performance computing, pointer-chasing (multiple dependent memory loads) is a known bottleneck that leads to many stalls [4]. Our approach avoids that by favoring contiguous array indexing.

Comparison to C++ approach: In C++, one might have each node contain a direct pointer to its dependency's value (e.g., a pointer to a `double` in the dependent node). In Rust, we can achieve the *effect* of that in two ways: 1. **Using indices (safe)** – as described, store `usize` indices for inputs and use them to index into a `values` array. This is fully safe and idiomatic, and with optimizations it's very close to pointer

performance. The minor overhead is the bounds check on each access (which can often be optimized out or mitigated by using `.get_unchecked` in an unsafe block if we are confident in validity). 2. **Using raw pointers (unsafe)** – after building the graph and allocating the `values` array, we can traverse the node list and replace each input index with a raw pointer to the corresponding entry in `values`. For example, a node might store `in1: *const f64` and `in2: *const f64` to its two inputs' values. Similarly, we could store a pointer to its own output location (or simply use its index as the output location). Then evaluation becomes:

```
for node in nodes.iter() {
    unsafe {
        match node.op {
            Op::Add => *node.out = *node.in1 + *node.in2,
            Op::Mul => *node.out = *node.in1 * *node.in2,
            // ...
        }
    }
}
```

This is essentially what the C++ version would do – directly dereferencing pointers to get values. It eliminates even the index-to-pointer calculation in the inner loop (we pay that cost just once upfront when setting up the pointers). In Rust, raw pointers and dereferencing are unsafe, but we've ensured they point into a valid `values` array and remain valid for the lifetime of evaluation. This approach is about as fast as it gets in Rust, equivalent to C++ pointer dereferences. It meets the requirement of *no hash tables, no strings, no extra lookups at runtime*. All we do in the hot loop is pointer dereferences and arithmetic.

Either approach (indices or cached pointers) ensures that at evaluation time we have **direct** access to dependency values with minimal overhead. We choose based on how much we want to lean on `unsafe`. Even the safe index approach, due to Rust's optimizations, can be very close in performance to raw pointers if done carefully.

Additional Considerations

- **Memory layout and cache:** By storing nodes (as structs containing their op enum and input references) in a `Vec`, they are contiguous in memory. Iterating over them is cache-friendly. Similarly, the values array is contiguous. This contiguous layout is in stark contrast to something like a linked structure or scattered `Box` allocations, and it leverages CPU caches much better [6]. We avoid memory reallocation during evaluation altogether. (If nodes are added incrementally, one should reserve capacity in the vectors to avoid reallocation, or build the final vectors once the graph structure is known.)
- **No runtime type checks beyond the enum match:** The enum match is effectively our "switch" on node type. Since it's a finite set, it's very efficient. There's no need for any string comparisons or dynamic type casting. This keeps the inner loop tight.
- **Parallelism (if needed):** The question didn't explicitly ask for it, but if performance needs are even higher and the graph has independent subgraphs, one could evaluate independent sections in parallel. The current design makes it easy to do so (for example, if there are multiple output nodes

that don't depend on each other, evaluate subgraphs on different threads). The key is ensuring no data races on the `values` array (which is doable if subgraphs write to separate portions of the array). However, if the graph is inherently sequential (each node depends on previous), parallelism won't help. In any case, our design is primarily about maximizing single-thread speed first.
- **Maintaining the enum:** The requirement was that adding a new node type shouldn't require "other crap to update" beyond one enum. Using a single `NodeOp` enum meets that – you add a variant and handle it in the match. There's no need to, say, register the node type in a separate factory or update multiple structures. This keeps maintenance manageable. The downside is the match function grows with new variants, but that is straightforward and usually fine.

Conclusion

By using a single enum for node operations and storing graph data in contiguous arrays, we achieve an evaluation loop that is as efficient as possible in Rust. We've eliminated dynamic dispatch and heap fragmentation, and we perform direct memory accesses for inputs. There are no hash table lookups or string matching at runtime – everything is resolved to direct indices/pointers ahead of time. This design essentially replicates the C++ approach of direct pointer references to dependency values, within Rust's constraints. It ensures that when evaluating the graph, the CPU is doing the bare minimum: load a few values, perform an arithmetic operation, store the result, then move to the next node. All the heavy lifting (figuring out dependencies, offsets, etc.) is done beforehand.

This approach is necessary because, as the user observed, if we handled inputs via strings, maps, or other indirections during the computation, it would be far too slow given the tiny amount of work each node does. Instead, our solution yields a tight, cache-friendly loop. In summary, we **can** achieve C++-like performance in Rust for this computation graph by careful structuring of data and code – avoiding all the overhead (dynamic dispatch, unnecessary data structures) and focusing on direct access and static dispatch [3] [1]. The end result should meet the very high performance requirements, evaluating even large graphs of simple arithmetic nodes with minimal latency.

Sources:

- Paul Kernfeld, *Exploring Computation Graphs in Rust* – discusses enum vs trait approaches; notes that dynamic dispatch has a performance cost compared to static dispatch [1].
- Reddit discussion on Rust dispatch – confirms that using an enum (static dispatch) can be considerably faster than trait object calls in hot loops [2].
- Rust Programming Language Forum – comment on using indices instead of references for graph nodes, and the pain of extra indirection in lookup tables [3]. This supports using a direct index/pointer approach for efficiency.
- Martin Thompson's *Top 10 Performance Mistakes* – highlights how not all memory access is equal and data-dependent memory access (e.g., pointer chasing or hash lookups) can severely impact performance [6]. Our design avoids such accesses.
- Research on irregular memory access patterns – emphasizes that data-dependent loads (pointer chasing in graphs) and unpredictable branches cause stalls and poor performance [4]. Our contiguous, predictable access pattern mitigates this issue.

---

1 Exploring Computation Graphs in Rust - Paul Kernfeld dot com
https://paulkernfeld.com/2018/06/17/exploring-computation-graphs-in-rust.html

2 5 Performance: Pattern matching vs Dynamic Dispatch : r/rust
https://www.reddit.com/r/rust/comments/1cx7qvi/performance_pattern_matching_vs_dynamic_dispatch/

3 Exploring computation graphs in Rust - The Rust Programming Language Forum
https://users.rust-lang.org/t/exploring-computation-graphs-in-rust/18139

4 tnm.engin.umich.edu
http://tnm.engin.umich.edu/wp-content/uploads/sites/353/2021/01/2021.02.Prodigy_HPCA2021_Camera_Ready.pdf

6 Top 10 Performance Mistakes - InfoQ
https://www.infoq.com/articles/top-10-performance-mistakes/